First publ. in: IEEE Computer Graphics and Applications, 19 (1999), 1, pp. 56-65

Interactive Modeling of Plants

Bernd Lintermann
Center for Media Arts and Technology Karlsruhe

Oliver Deussen
Otto-von-Guericke University of Magdeburg

We see plants nearly everywhere in our environment. They dominate outdoor scenes and most interior scenes as well. So why do we only have a few satisfactory plant models? We think it's because, so far, creating plants is a job for experts who can handle the large structural and geometrical complexity of these models.

Combining a rule-based approach with traditional geometric modeling techniques enables the fast and comfortable generation of plants.

In this article we present a modeling method that allows easy generation of many branching objects including flowers, bushes, trees, and even nonbotanical things. A set of components describing structural and geometrical elements of plants maps to a graph that forms the description of a specific plant and generates the geometry. Users get immediate feedback on what they've created—geometrical parameters, tropisms, and free-form deformations can control the overall shape of a plant. We'll demonstrate that our method handles the complexity of most real plants.

Before we discuss related work on interactive aspects of plant modeling systems, we want to stress that plant modeling aims to achieve two distinct goals. One is biologically motivated—people try to simulate the development of natural plants. The other seeks to generate only visually correct shapes of plants. This reflects the need for good geometric models in many computer graphics applications. Our method focuses on the second goal and tries to give users as much modeling power as possible for creating different plants.

In the beginning, work on plant generation was biologically motivated. Pioneering work by Lindenmayer and later by Prusinkiewicz described the structure of plants by string rewriting systems (L-systems) operating on a set of rules.[1] The approach includes context sensitivity as well as stochastic behavior and was recently extended to let plants adapt to environmental effects.[2]

L-systems specify plants in terms of local growth rules. This may be intuitive for biologists, but from the modeling point of view we're interested in dealing directly with the global characteristics of plants. Lindenmayer and Prusinkiewicz created a virtual laboratory for L-systems that lets users customize the models by specifying simulation parameters graphically.[1] However, controlling the global aspects of the shape directly still proves difficult. Also, users get no direct feedback because they must rerun the simulation after changing the parameters.

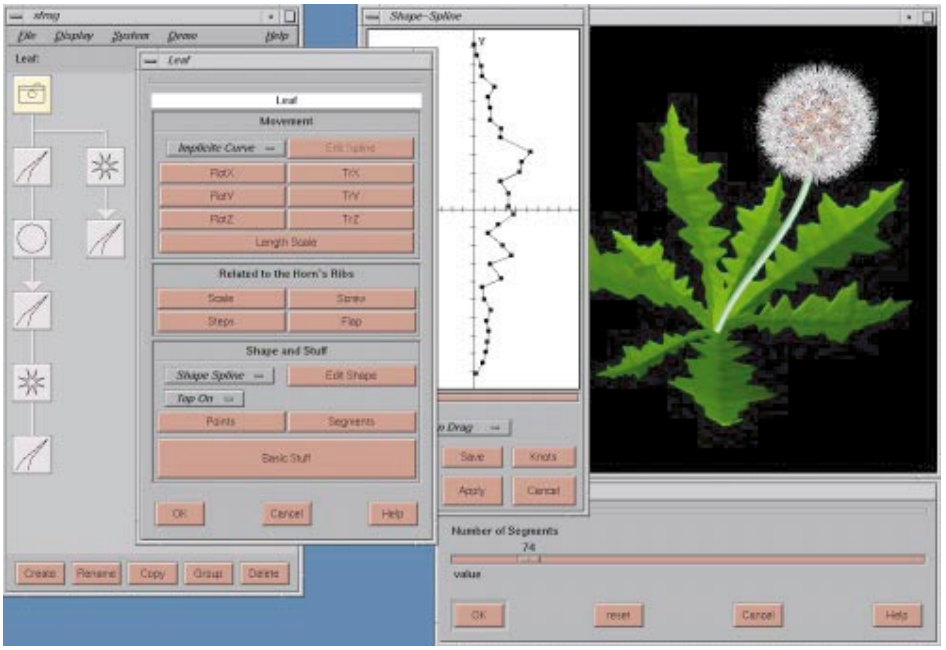
To provide more intuitive parameters and to ease control over the models, others have developed customizable procedural plant models. Oppenheimer[3] presented a fractal tree model where, in each branch, users can specify parameters like branching angle, the size ratio between the main stem and branch, or the number of branches per stem.

De Reffye et al.[4] developed a procedural model based on the birth and death of growing buds that lets users control the generation of plants by some parameters. AMAP, a commercial library of plants (particularly trees) and generation procedures, builds on this idea (see http://www.cirad.fr/amap/amap.html). The user edits a plant type's parameters and runs the simulation to produce the desired geometry.

Holton's[5] procedural model of trees assigns a strand to any path from the root to a tree's leaves. The number of strands in a fork determines the branches' fork angle, length, and taper. This lets the models be parameterized on the level of various tropisms that control growth. After specifying the parameters, the models are rendered—a time-consuming process that makes interaction difficult.

While the authors listed above concentrated on finding efficient descriptions that fit into botanical principles, Weber and Penn[6] focused on the second goal of generating a visually favorable geometry without adhering strictly to botanical laws—they put special emphasis on modeling the overall shape of a tree. Specifying the shape geometrically and restricting the model to grow within the bounds of the shape accomplished this. A set of textually edited parameters described the geometry for each branching level of the tree.

Onyx Computing's TreeMaker (http://www.onyxtree.



1 The modeler's main dialog windows. The plant's structural description is modeled in the upper left, and the results appear on the right. The parameters of each component can be changed with special dialogs. Here the outline of a leaf is created, as shown in the center shape-spline box.

com), a highly interactive system for generating trees, lets users customize a tree's branching levels graphically. For example, users can cut a stem and receive immediate feedback on the newly created geometry. This system makes it easy to generate and modify the models, but is restricted to trees.

We aimed to combine the power of a rule-based approach with the intuitiveness of generic tree methods. Also, we wanted to have a general modeling technique that could generate nearly all kinds of plants by using one consistent description. For modeling purposes, we also needed a highly interactive system with direct feedback.

Our solution uses a graph description. The nodes of the graph are components that represent parts of a plant, and the edges denote creation dependencies. Hart[7] used a similar graph to describe fractal geometries and a limited class of L-systems. In his approach, the nodes describe instances of geometrical primitives and the edges also denote creation dependencies. Hart's system traverses the graph to produce the geometry.

In our approach,[8] we divide the generation of geometry into two steps (see http://www.greenworks.de). First, we expand the graph to a tree. We do this because structural information is represented by the graph structure and by a special class of components that multiplies its children algorithmically. Second, our system traverses the tree to produce the geometry.

Modeling with components

In our approach, components encapsulate data and algorithms for generating plant elements. Generally three categories exist: one group of components creates graphical objects like stems, twigs, leaves, or geometric primitives; the second multiplies other components; and the third applies global modeling techniques. All components own a set of parameters to control their behavior.

Our technique builds on the idea that graph-based systems generate structures powerfully. If a graph generates plant parts represented by components, then we can conserve this power and combine it with individually optimized algorithms for generating and multiplying geometry. Also, the description proves much more intuitive because it consists of high-level units only.

Within each component, the procedural modeling method specifies its behavior. A graphical dialog can be used at this point to optimize the interaction (see Figure 1). For example, our system's Leaf component uses polygons to specify a leaf's outline. Applying values or editing a spline curve lets users define the leaf's lateral and longitudinal curvature.

In many cases, objects must be distributed algorithmically. For instance, placing objects on a surface according to the golden section (such as distributing seeds on a sunflower) is a classical example for this need. Multiplying components, the second component group, helps achieve such an algorithmic distribution. One parameter of this component is the number of multiplied components; others control their placement and orientation.

We can now establish the complete plant description by connecting component prototypes in a directed graph called the prototype graph, or *p-graph*. When our system traverses the p-graph, it builds a temporary tree of component instances that we call the *i-tree*. This tree is used to generate the geometry.

Each link between two component prototypes in the p-graph denotes a creation dependency. If the source prototype was copied to create an instance, the system does the same thing with the target prototype and connects both instances by a link.

The main difference between classical rule-based approaches and the graph-based object instancing paradigm[7] is that structural information is represented in two ways. One representation is given by the p-graph's links, and the other is specified by the parameters of multiplying components.

If a multiplying component is part of the graph, during the expansion the system creates as many instances of all subsequent components as defined in the multiplying component. All instances of the child components link to the instance of the multiplier.

To obtain children that differ in geometry, the prototype of a multiplying component stores several parameters (for example, the size of its children) as ranges. The system interpolates values to obtain the specific value for each instance of a child. If the parameter range is $[v_0, v_1]$, the value of the i-th child out of n is $v_i = v_0 + i(v_1 - v_0)/(n-1)$. Before the system uses this parameter value, it applies an arbitrary function that the user specifies. This lets us introduce randomness to the models and vary the shape individually for different components.

Each component prototype has a parameter defining the highest recursion depth. If defined within the p-graph, a recursion transforms into a subtree by producing instances that depend on the recursion depth.

Now all the instances form the i-tree. The i-tree's root component is enforced to produce its output, and it forces all its children to do the same. This proceeds until the system has traversed the whole tree.

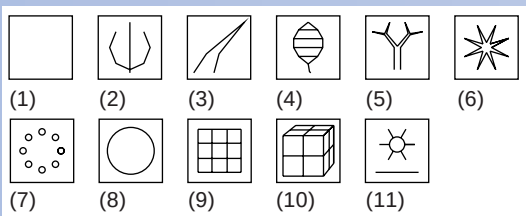
A geometry generation example

A short example may clarify the process. Component prototype A is the root. Prototype B is a geometry-producing component and should generate a stem. C is a multiplying component that should produce three instances of its children and no geometry. Prototype D produces part of a twig and has a recursion depth of three. X_i denotes the instances of component X.

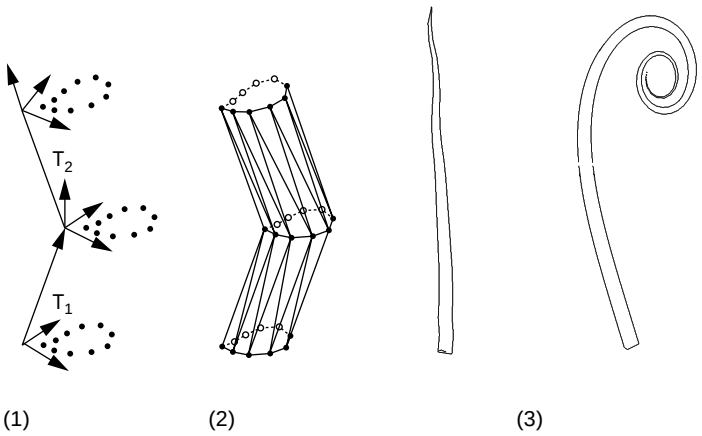
Figure 2a shows the components (A to D) that form the p-graph, and Figure 2b shows the resulting i-tree. Component prototype C generates three instances of D, namely D_1, D_2, and D_3, according to its local multiplication parameter and connects C_1 to them. The recursion defined on prototype D forms a sequence of three

Component Types of the Modeler

Here are the different component types that our modeler uses. Figure A shows the icons for each component.



A The different component types of our modeler: (1) Simple, (2) Revo, (3) Horn, (4) Leaf, (5) Tree, (6) Hydra, (7) Wreath, (8) Phiball, (9) FFD, (10) Hyperpatch, and (11) World.



B Geometry definition used by a Horn component: (1) point sets are defined relative to preceding sets; (2) the triangulation forms the outline; (3) two examples of a Horn geometry.

Simple

All components offer a basic set of parameters that define a geometric primitive and transform the data of child components in the p-graph. The Simple component has only this basic parameter set, and it creates simple geometries.

Revo

Revo, another kind of geometric primitive, produces a surface of revolution. In addition to the basic parameters, the user can specify the outline of the surface of revolution.

Horn

The geometry produced by the Horn component forms the basis for all kinds of stems or twigs. Oppenheimer[3] as well as Todd and Latham[11] did this similarly: a sequence of primitives is generated either along a spline or by defining relative transformations between primitives.

If the primitives are selected as tubes, the Horn creates a sequence of point sets. These are interpreted as cross-sections of a closed volume and are triangulated later. In this case, the geometry forms a generalized cylinder. Figure B shows the process and gives two examples.

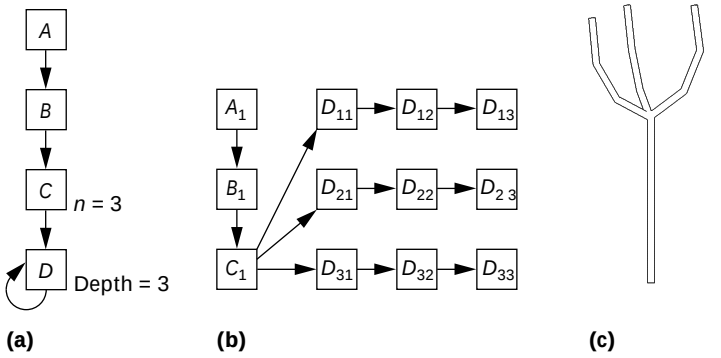
Other components like the Simple component can also produce single point sets. The point sets generated by subsequent instances in the i-tree are triangulated. This lets us achieve a wide variety of surfaces. The parameters of the Horn component specify the shape, direction, and length of the generated geometry.

Leaf

Leaf components define natural-looking leaves and petals. Similar to Horn components, they produce sequences of primitives or point sets. In this case, the point sets are open and define a

instances—D_{i1}, D_{i2}, and D_{i3}. This causes nine instances of D to take place in the final tree. Figure 2b shows the i-tree, and Figure 2c shows the geometrical output.

Local coordinate systems—calculated separately for each child instance of a multiplication component—produce the differences in the orientation of the nine instances' geometries. Additionally, each component executes a geometry generation method triggered by its parameters.



2 To generate a plant, the user constructs the p-graph (a), which expands to form the desired i-tree (b). The tree then produces the desired geometry (c).

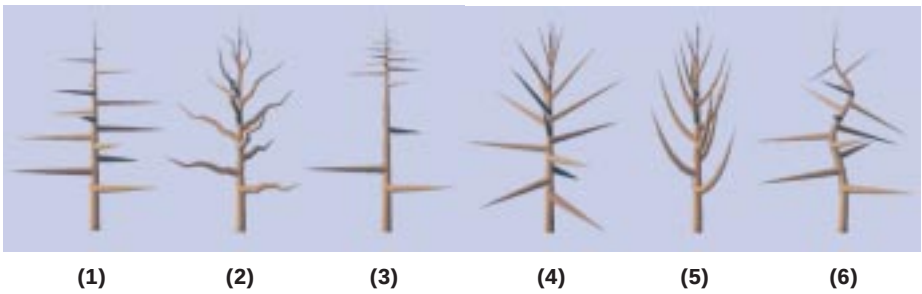
Component types and their combination

Here we'll focus on the different kinds of components and their parameters. The sidebar "Component Types of the Modeler" explains all the system's components. As mentioned previously, we distinguish among three component categories: geometry generating, multiplying, and global modeling.

Each component offers a basic set of parameters that defines a geometric primitive and transforms the child surface. The main difference to the Horn component is the method affecting the generated geometry. Several deformations can be applied and the leaf's outline can be specified as a polygonal curve.

It's also possible to apply **a** phototropism. The scalar value determines how much the local coordinate system of the leaf should be rotated in order to lie perpendicular to a given global light field vector (defined by the World component listed below).

Practical experience showed us that using textured leaves is important for all kind of plants. If we use a texture, it's projected on the geometry produced by the Leaf component.



C Parameter variations in a tree component: (1) default outline; (2) gnarled branches; (3) branching density; (4) branching angle; (5) phototropism; and (6) stem curvature.

Tree

The Tree component is a mixture between geometry-producing and multiplying components. Like the Horn component, it creates a horn-like geometry by default. The difference to the Horn component lies in how subsequent components in the p-graph are treated. The Tree component can be instructed to multiply them as branches of its stem. This lets the user define a whole tree by a cascade of Tree components or by a recursive definition.

In contrast to L-systems, which work by using local growth rules, this mechanism lets users modify global characteristics of the plant directly. For example, the branching angle controls the angle of branches along the whole stem (Figure C4). It's easy to change the angle of the lower branches only, if desired.

The other parameters control the density of branches along the stem, the deviation of the stem away from its main axis, the shape of the stem, the phototropism or gravitropism to apply, or the outline (curvature) of the stem.

The system generates the shape, curvature, deviation, gnarled look, and tropisms of the stem by changing the positions of the triangulated point sets used for geometry definition (Figure B). The curvature parameter defines the direction of the stem directly, while deviation changes the direction of the stem if a branch is generated. Tropisms (in Figure C5 applied to the twigs) are generated by changing the local point sets of the tubes (Figure B1) in order to lie perpendicular to a given light or gravitational field vector (see World component below).

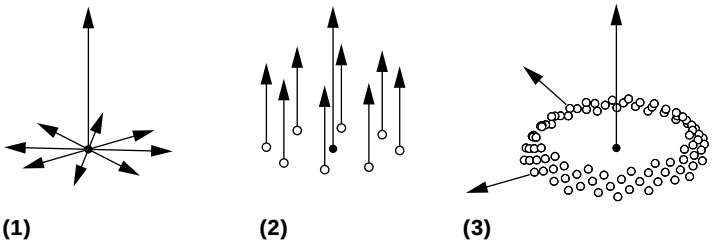
Hydra and Wreath

The Hydra component multiplies subsequent components of the p-graph and places them in a

continued on p. 6

continued from p. 5

star-like arrangement. The Wreath component arranges multiplied children similar to candles on a Christmas wreath. Figures D1 and D2 show placement and orientation of the multiplied components. Both components specify the number of generated children. Additionally, the user can edit the radius as well as the opening and closing angle of the circle.



D Placement and orientation of multiplied components: (1) multiplication by a Hydra component; (2) by a Wreath component; and (3) by a Phiball component.

Phiball

Components multiplied by a Phiball component are arranged on a section of a sphere by the golden section algorithm. Instead of using a collision-based model,[10,11] we found an analytical solution for multiplying components on a spherical section. We describe the calculation in detail in the sidebar "Golden Section Placement on a Sphere."

Given parameters include the number of multiplied children, the radius and the opening and closing angle of the spherical section, and the size of the children and their influence on the placement.

FFD and Hyperpatch

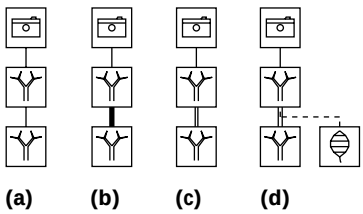
Both component types define a FFD on a portion of the produced geometric data. The system supplies several methods for defining FFDs. While the FFD component specifies free-form deformations by functions, the Hyperpatch component does this by moving control points of a 3D Bézier function.

FFD components can also switch off the influence of preceding free-form deformations. This makes it possible to deform stems and twigs of a tree but not the leaves. One parameter of the FFD component can switch between deforming all the triangles generated by subsequent components or just the skeleton. Also, users can specify the degree of the 3D Bézier patch (linear to quartic).

World

Tree and Leaf components can be forced to produce geometry according to gravitropism and phototropism. By default, the corresponding field vectors are defined in the positive and negative *z*-direction. By using a World component, these definitions can be substituted by arbitrary functions.

3 Methods in combining components: (a) child link (thin line); (b) branch link (bold line); (c) recursive combination (double line); and (d) recursive combination with additional leaf link (dashed line).



components' data in the p-graph. One parameter is the geometrical primitive. Beneath standard primitives like cubes, balls, and so on, we implemented two special types of primitives, called Area and Tube. They define open and closed point sets that will be triangulated in a later step. Other parameters concern the transformation, color and texture of subsequent geometry, and recursion depth. The Simple component—one of the five entities that form the geometry-producing components—offers only this basic parameter set. Others handle more complex tasks: Revo for creating surfaces of revolution, Horn for twigs and stems, Leaf for leaves, and Tree for modeling trees.

The Tree component also offers multiplying functionality because twigs can be multiplied around the stem. Other multipliers are Wreath, Hydra, and Phiball. Three components handle global modeling and form the last group of components: FFD (free-form deformation), Hyperpatch, and World, which all affect the generated plant's overall shape.

These components combine to form p-graphs (see Figure 3) of various models. Each p-graph has an additional root component, represented by a camera icon. Parameters of this component control the view and positions of light sources.

During the creation of a p-graph, the user chooses components from a graphical toolbox, which are automatically placed next to the graph. Selecting a component and moving it onto another component establishes a link. Three types of links can be used, as follows:

- *Child link*: The standard link. The component's geometry is placed relative to the preceding component's geometry. The p-graph displays these links as thin lines.
- *Branch link*: The child component is multiplied as a branch if the parent is a Tree component. In all other cases it's treated as child link. These links appear as bold lines.
- *Leaf link*: If the parent component is part of a recursion, the child is created only once after the recursion terminates (the target is a leaf in the structural sense). These links appear as dashed lines.

Golden Section Placement on a Sphere

To derive the mechanism for a spherical placement according to the golden section, we first assume the distributed objects have the same size. The sphere has a rotation axis **z**. We want to place N objects around the **z**-axis with angle $\Phi_i = i \times d\Phi$ for the i-th object and $d\Phi = 2\pi/((1+\sqrt{5})/2)$, the golden section. Each object is also rotated around an axis perpendicular to the **z**-axis by angle θ_i, which will be determined below. First, we compute the height of a spherical section that provides space for N objects with an area A_E. On a sphere with radius R, the section $S(\theta_1, \theta_2)$, $0 \le \theta_1 < \theta_2$ occupies an area of $A = 2\pi Rh$ with $h = R(\cos(\theta_1) - \cos(\theta_2))$.

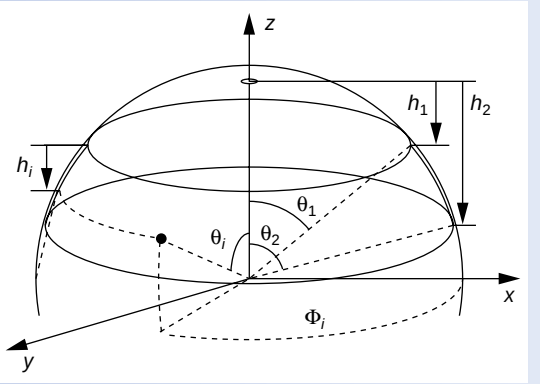
To place N objects with an area $A_N = N A_E$, a sphere with radius

$$R = \sqrt{\frac{A_N}{2\pi(\cos(\theta_1) - \cos(\theta_2))}}$$

is needed. For computing θ_i within the i-th iteration, we express the area occupied by i objects $A_i = iA_E$ as a function of the section's height. We make no assumption about the shape of the objects and assume that they can be packed tight, which is generally not the case if all objects have the same shape. In this case, the uncovered area has to be incorporated into A_i heuristically.

$$A_i = 2\pi R^2 l_i \text{ with } l_i = \frac{A_i}{2\pi R^2} = \frac{A_i h}{A_N}$$

Now we compute the angle θ_i by



E Placing points on a sphere by the golden section.

$$\theta_i = \arcsin\left(\frac{R - (h_1 + l_i)}{R}\right) = \arcsin\left(\frac{R - R(1 - \cos(\theta_1)) - \frac{A_i h}{A_N}}{R}\right) = \arcsin(\cos(\theta_1) - \frac{A_i}{A_N}(\cos(\theta_1) - \cos(\theta_2)))$$

If objects vary in size, let a_j be the area of the object j. In this case the variables A_i and A_N of the equation above are computed by

$$A_N = \sum_{j=1}^{N} a_j \text{ and } A_i = \sum_{j=1}^{i} a_j$$

For objects with constant size, we determine the angle by

$$\theta_i = \arcsin(\cos(\theta_1) - \frac{i}{N}(\cos(\theta_1) - \cos(\theta_2)))$$

In the system's display the arrows are omitted denoting the direction of the links in Figure 3 because all links point away from the camera icon. In addition, links that are part of a recursion are drawn as double lines and the back-pointing links are omitted. The system duplicated and appended the first component to the end of the recursive part. This makes it possible to display the p-graph as a tree (see Figures 3c and 3d), which avoids many graph drawing problems and is visually more pleasant.

Some examples

Now that we've described components and parameters as well as their combinations, we'll give three examples of how to model plants. A flower, a bush, and a whole tree demonstrate the generality of our approach.

Sunflower

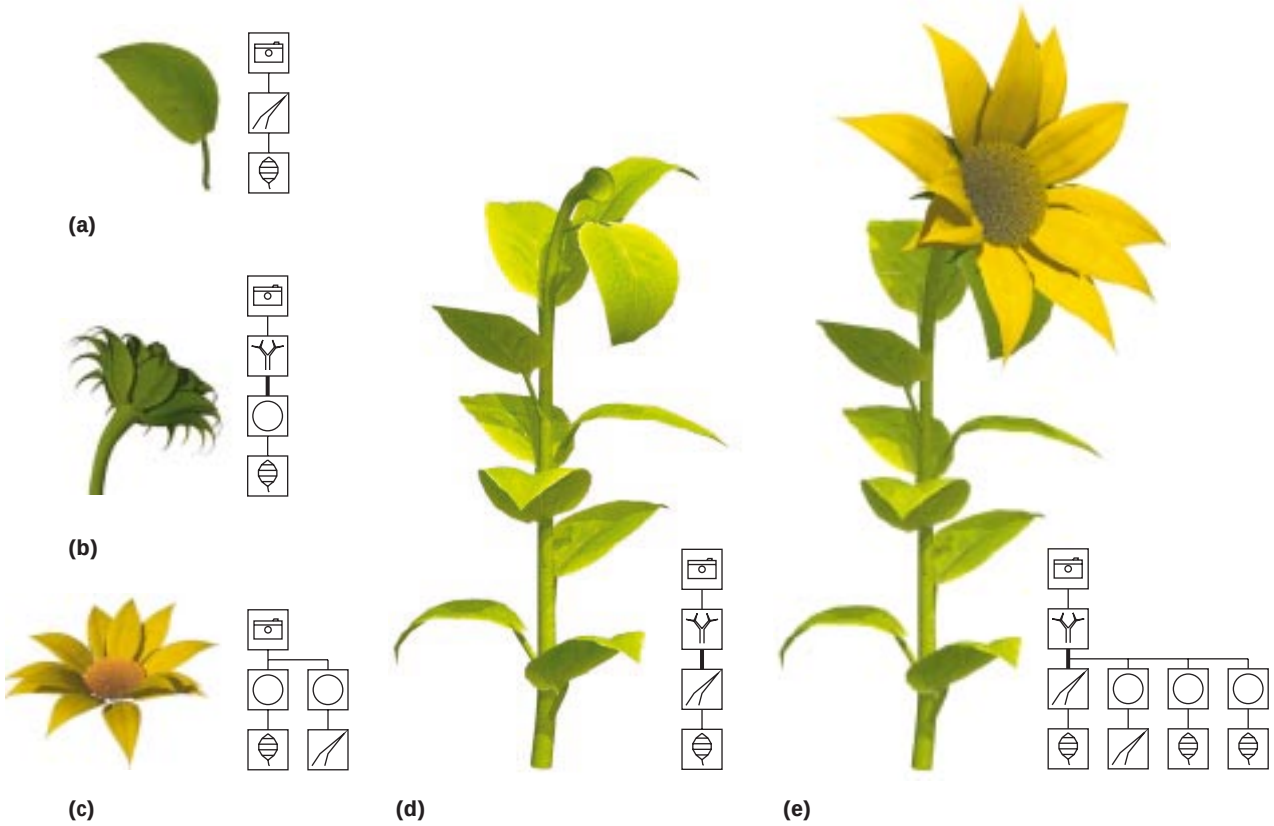
First, we'll model a small sunflower. A sunflower's natural leaf is scanned and applied as a texture to the Leaf component's surface (see Figure 4a, next page). A tiny stalk is connected to the leaf. So far, the p-graph consists of three components—the camera, a Horn for the stalk, and a Leaf. By editing splines, users can choose the appropriate curvature and scale of the Horn and Leaf to create a typical outline of a small leaf.

Next, the system iterates the leaves as branches of a Tree describing the plant's stalk (Figure 4d). The top of the stalk is opened to form the head of the flower. Figure 4b shows this process, where a Phiball iterates some leaves and places them on the top of the stalk (here the thin line between the Tree and Phiball indicates that it's a child link).

Similarly, two Phiball components construct the blossom of the sunflower—one for arranging the petals and the other for arranging the seeds (Figure 4c). Finally, everything is connected to the full p-graph (Figure 4e).

Rhododendron

A rhododendron is a good example for modeling a medium-sized bush. Here the geometric complexity increases. The focus is on the branching structure rather than on geometric properties of leaves and blossoms.



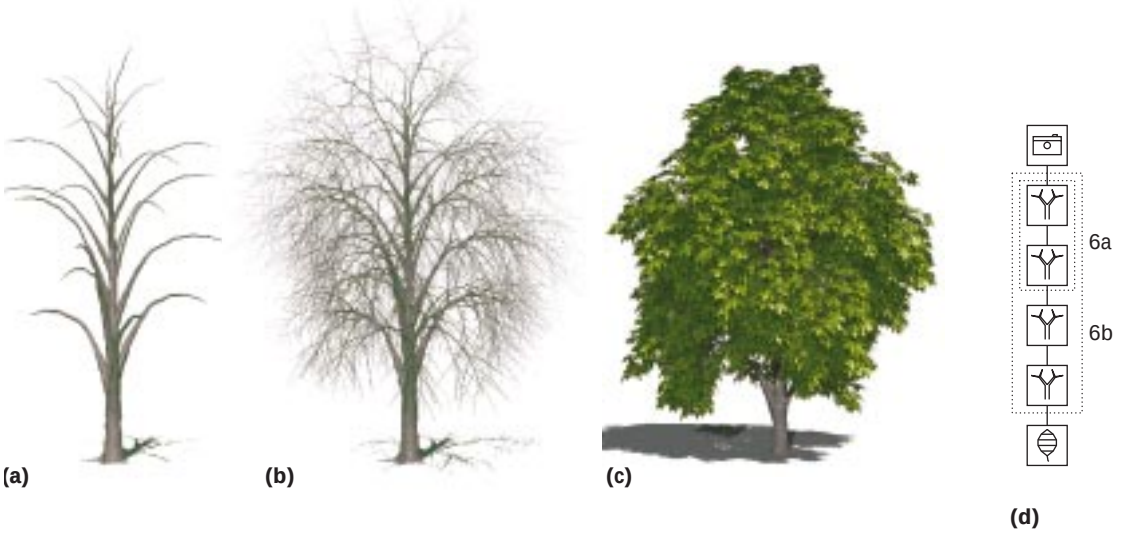
4 Parts of a sunflower with corresponding p-graphs.

Again, we'll first create a single leaf by scanning a natural leaf and using it in the Leaf component (see Figure 5a; Figure 5e shows the complete p-graph).

Next, the system creates a twig. It builds the top of the twig separately since the Rhododendron's leaves are specially arranged around the blossoms. We use a Phiball to multiply the leaves and place a blossom inside (Figure 5b). At this point, the p-graph consists of six components (these are inside the dotted region of Figure 5e where 5b is referenced). A Tree multiplies the leaves around the stalk, and two additional components are connected to the Tree component by child links: a Phiball for multiplying the leaves around the blossom and a Hydra for the petals. This component arranges the blossom's leaves. The corresponding Leaf is the child of the second Phiball. Also, the normal leaves are connected to the first Phiball as children.

Constructing the whole twig presents a fundamental problem. The twig branches for leaves and tiny twigs, but we don't want to have a twig at every leaf position and vice versa. Similar problems arise when we want to model exceptions—for example, a tree with some dead

5 Modeling a rhododendron: (a) Leaf with texture; (b) a tiny twig with blossom; (c) a main twig that branches in leaves and in two tiny twigs; (d) the whole bush, and (e) p-graphs of the rhododendron (the numbers of the dashed regions indicate the figures corresponding to the subgraphs).



6 **A tree modeled by a sequence of Tree components. (a) First, two components are combined and the parameters are adjusted. (b) Two more branching levels are constructed. (c) Adding the leaves yields the final tree. (d) P-graphs of the chestnut tree (the numbers of the dashed regions indicate the figures corresponding to the subgraphs).**

twigs or a flower with some irregularly formed leaves.

We solved this problem by introducing a list of flags for each component. Every multiplied component receives a parameter value from the multiplier that indicates its iteration number. The list indicates if an iteration number should be created by a multiplying component or not. If two subgraphs are connected to a multiplier, and one list forces the creation of components without the numbers 2 and 13, and the other list has the complementary set (in this case the iteration numbers 2 and 13), we get the desired result. This is exactly how we created a twig for the rhododendron, which branches for two tiny twigs and some dozen leaves (see Figure 5c).

The last step is to arrange some of these twigs (we used 20) around the center of the bush. Adding a Phiball component to the root of our p-graph accomplishes this, although the twigs must be scaled a bit to achieve the overall shape of a rhododendron. Figure 5d shows the final image (we also created the container with the modeler).

Chestnut tree

The overall p-graph structure of most trees we've modeled so far is quite uniform. A cascade of Tree components represents the branching structure. Leaves, needles, or blossoms multiply as branches of the last Tree component.

To model a chestnut tree, we again use a scanned natural leaf to get the right texture for our Leaf component. This has the advantage that just a few triangles can create the geometry of a single leaf. This is an important fact, since a full tree may have up to several hundreds of thousands of leaves.

By combining two Tree components we get a simple tree. Changes in the parameters for the number of branches, the spacing of the branches, the branching angle, and shape lead to a more naturalistic tree (see Figure 6a). Now, we add two more Tree components to model the tiny twigs (Figure 6b). Finally, we add the leaves and apply a texture to the stem (Figure 6c).

7 **Experienced users can model very complex trees, such as the one shown here, in a few hours.**

Though it sounds very simple, modeling a real tree is difficult. An experienced user takes several hours to model a tree from scratch. Nevertheless, our system is very fast compared to other modeling methods. For example, the very complex maple tree shown in Figure 7 was modeled in five hours. What's more, models can easily be reused and changed in order to get new tree models.

Shape modeling

Often a scene requires more than just modeling a plant—sometimes it calls for specific shaping, such as a tree growing beneath a wall or a plant that's partially

8 The curvature of the leaves can be defined by depending on the iteration number and a random function.

(a) (b)

9 Two examples of tropisms: (a) a weeping willow with a strong gravitropism and (b) a philodendron growing around a stick.

shadowed or moved by the wind. Experiences with our approach showed us that just four additional modeling methods suffice to generate a wide variety of shapes and desired artifacts.

We've already mentioned one method: modeling of exceptions with a list of flags. Some kind of context sensitivity must be introduced here, because the multiplier specifies a context by generating the iteration number indicating if a multiplied component has been generated or not.

The three other modeling features are functional modeling, tropisms, and free-form deformations, which we'll discuss next.

Functional modeling

Our system can introduce randomness and other functionally specified properties to our models. Wherever a component prototype uses a parameter interval—for example, the scaling of multiplied components or the curvature of a leaf—a function can be applied before the system uses the interpolated parameters of that interval.

Standard mathematical functions, like sine, cosine, and so on can be applied, but so can a random function. Parameters such as recursion depth or iteration number can be used inside these functions. The user can also define arbitrary functions for each parameter interval. Figure 8 shows an agave that demonstrates this effect. The almost vertical leaves are less curved than the others. We did this by using the iteration number that the Phiball component set during multiplying. Adding a random offset helps achieve a natural appearance.

Tropisms

We mentioned Tropisms when we discussed the parameters of the Tree and World components. They specify the sensitivity of a plant or a part of a plant to the global influence of gravity and light direction.

Tropisms can model some very different effects. While we used a traditional gravitropism to create the weeping willow in Figure 9a, we introduced a cylindrical tropism for modeling a philodendron around a stick in Figure 9b. Similarly, the influence of wind can be simulated by a horizontal tropism, or a plant can be forced to grow along a wall.

Free-form deformations

We can use free-form deformations (FFDs) as another way to change the entire model's shape. As mentioned previously, the Hyperpatch component uses a 3D cube for that purpose. The user selects one or more points and moves them parallel to the viewing plane. The points are control points of a 3D Bézier function, which defines the desired deformation.

Often, only parts of the model should be deformed. For example, the twists of a tree should be deformed, but not the leaves or needles. Placing another FFD or Hyperpatch component in the subtree affected by a free-

(a) (b) (c)

10 Free-form deformation applied to a pine: (a) undeformed model; (b) deformation of the twigs (the needles remain unchanged); and (c) deformation of the whole tree.

form deformation accomplishes this. These components separate the deformations defined above.

Figure 10 shows an example of a partial free-form deformation. In Figure 10b the twists of the pine are deformed, but the needles remain unchanged. Figure 10c shows the result of deforming the whole tree. Here, the needles on the right appear too big—a result of the strong deformation.

Conclusion

We presented a new and powerful method for modeling plants. In contrast to previous approaches our method lets users design a wide variety of plants by mostly intuitive mechanisms.

Nearly all parameters can be changed graphically. We evaluated[8] the user interface and tested the modeler by constructing large models with millions of polygons and very complex structure. Recently, the modeler was used in a project that dealt with generating whole ecosystems.[12]

In future work we'll focus on a better interface for some parameters and the implementation of other multiplication components. Also, we'll extend the set of existing models. We must also work on a more detailed analysis of our method in comparison to other plant-generating methods. ■

Acknowledgments

Many thanks to Przemyslaw Prusinkiewicz for his constructive suggestions regarding this article. We also thank Alfred Schmitt (University of Karlsruhe), Jeffrey Shaw (Center for Media Arts and Technology, Karlsruhe), and Thomas Strothotte (University of Magdeburg) for supporting our work. Sylvia Zabel and Michiel Smid (University of Magdeburg) helped proofread this article.

References

1. P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*, Springer-Verlag, New York, 1990.
2. R. Méch and P. Prusinkiewicz, "Visual Models of Plants Interacting with their Environment," *Computer Graphics* (Proc. Siggraph 96), ACM Press, New York, 1996, pp. 397-410.
3. P.E. Oppenheimer, "Real-Time Design and Animation of Fractal Plants and Trees," *Computer Graphics* (Proc. Siggraph 86), Vol. 20, ACM Press, New York, 1986, pp. 55-64.
4. P. de Reffye et al., "Plant Models Faithful to Botanical Structure and Development," *Computer Graphics* (Proc. Siggraph 88), Vol. 22, ACM Press, New York, 1988, pp. 151-158.
5. M. Holton, "Strands, Gravity, and Botanical Tree Imagery," *Computer Graphics Forum*, Vol. 13, No. 1, 1994, pp. 57-67.
6. J. Weber and J. Penn, "Creation and Rendering of Realistic Trees," *Computer Graphics* (Proc. Siggraph 95), ACM Press, New York, 1995, pp. 119-128.
7. J. Hart, "The Object Instancing Paradigm for Linear Fractal Modeling," *Proc. Graphics Interface 92*, Morgan Kaufmann, San Francisco, 1992, pp. 224-231.
8. O. Deussen and B. Lintermann, "A Modeling Method and User Interface for Creating Plants," *Proc. Graphics Interface 97*, Morgan Kaufmann, San Francisco, 1997, pp. 189-197.
9. S. Todd and W. Latham, *Evolutionary Art and Computers*, Academic Press, London, 1992.
10. D.R. Fowler, J. Hanan, and P. Prusinkiewicz, "Modeling Spiral Phyllotaxis," *Computers and Graphics*, Vol. 13, No. 3, 1989, pp. 291-296.
11. D.R. Fowler, P. Prusinkiewicz, and J. Battjes, "A Collision-Based Model of Spiral Phyllotaxis," *Computer Graphics* (Proc. Siggraph 92), Vol. 26, ACM Press, New York, 1992, pp. 361-368.
12. O. Deussen et al., "Realistic Modeling and Rendering of Plant Ecosystems," *Computer Graphics* (Proc. Siggraph 98), ACM Press, New York, 1998, pp. 275-286.

Bernd Lintermann *is a graphics programmer and artist at the Center for Art and Media Technology Karlsruhe (ZKM). His research interests include methods for modeling, animation, and genetic evolution of organic objects and their use in the arts. He created several media art works for international exhibitions.*

Oliver Deussen *is working on his habilitation at the Otto-von-Guericke University of Magdeburg. His research interests include the simulation, modeling, and visualizing of complex biological objects; nonphotorealistic rendering; human-computer interaction; and synthetic holography. He received his PhD in computer science from the University of Karlsruhe in 1996. He is a member of ACM Siggraph, IEEE, and Eurographics.*

Readers may contact Deussen at the Faculty of Computer Science, Department of Simulation and Graphics, Otto von Guericke University, D-39016 Magdeburg, Germany, e-mail deussen@isg.cs.uni-magdeburg.de, http://isgwww.cs.uni-magdeburg.de/~deussen.